Apellido y Nombre: _____

Carrera: _____ DNI: _____

[Llenar con letra mayúscula de imprenta GRANDE]

Universidad Nacional del Litoral
Facultad de Ingeniería y Ciencias Hídricas
Departamento de Informática

Algoritmos y Estructuras de Datos

Algoritmos y Estructuras de Datos.
Recuperatorio. [28 de Noviembre de 2008]

- **[CLASES-P1]**

  Escribir la implementación en `C++` del `TAD LISTA` (clase `list`) implementado por celdas (simple o doblemente) enlazadas por punteros . Las funciones a implementar son `insert(p,x)`, `erase(p)`, `next()`/`iterator::operator++(int)`, `list()`, `clear()`.

- **[CLASES-P2]** Escribir la implementación en `C++` del `TAD ARBOL ORDENADO ORIENTADO` (clase `tree`). Las funciones a implementar son `insert(p,x)`, `erase(p)`, `find(x)` y `clear()`.

- **[CLASES-P3]**

  1. Escribir una función
     `bool abb_erase(btree<int> &T,int x);`
     que elimina un elemento `x` de un ABB y retorna `true` si la eliminación fue exitosa y `false` en caso contrario. Si necesita una función auxiliar (por ejemplo `min()`) debe implementarla.
  2. Escribir una función
     `void set_intersection(vector<bool> &A,vector<bool> &B);`
     que realiza la operación $A \leftarrow A \cap B$, es decir **in-place**.

- **[PROG-P1]**

  1. **[nilpot]**. Dadas dos correspondencias M_1 y M_2 la *"composición"* de ambas es la correspondencia $M = M_2 \circ M_1$ tal que si $M_1[a] = b$ y $M_2[b] = c$, entonces $M[a] = c$. Por ejemplo, si `M1={(0,1),(1,2),(2,0),(3,4),(4,3)}`, y `M2={(0,1),(1,0),(2,3),(3,4),(4,2)}`, entonces $M = M_1 \circ M_2$ `={(0,0),(1,3),(2,1),(3,2),(4,4)}`. Notemos que para que sea posible componer las dos correspondencias es necesario que los valores del contradominio de M_1 estén incluidos en las claves de M_2. Si el conjunto de valores del contradominio de una correspondencia M está incluido en el conjunto de sus claves, entonces podemos componer a M consigo misma, es decir $M^2 = M \circ M$. Por ejemplo, $M_1^2 = M_1 \circ M_1$`= {(0,2),(1,0),(2,1),(3,3),(4,4)}`. De la misma manera puede definirse, $M^3, ..., M^n$, componiendo sucesivamente. Puede demostrarse que, para algún n debe ser $M^n = I$, donde I es la *"correspondencia identidad"*, es decir aquella tal que $I[x] = x$. Por ejemplo, si `M = {(0,1),(1,2),(2,0)}`, entonces para $n = 3$, $M^n = M^3 = I$.
     Consigna: Escribir una función `int nilpot(map<int,int> &M);` que dada una correspondencia `M` retorna el mínimo entero `n` tal que $M^n = I$.
     Sugerencia: Escribir dos funciones auxiliares:
     - `void compose(map<int,int> &M1,map<int,int> &M2,map<int,int> &M);` que dadas dos correspondencias `M1, M2`, calcula la composición $M = M_2 \circ M_1$, devolviéndola en el argumento `M`,
     - `bool is_identity(map<int,int> &M);` que dada una correspondencia `M`, retorna `true` si `M` es la identidad, y `false` en caso contrario.

2. [**apply-map**] Escribir una funcion
`void apply_map(list<int> &L, map<int,int> &M, list<int> &ML)` que, dada una lista `L` y una correspondencia `M` retorna por `ML` una lista con los resultados de aplicar `M` a los elementos de `L`. Si algún elemento de `L` no está en el dominio de `M` entonces el elemento correspondiente de `ML` no es incluido. Por ejemplo, si

$$\texttt{L} = (1, 2, 3, 4, 5, 6, 7, 1, 2, 3)$$
$$\texttt{M} = \{(1, 2), (2, 3), (3, 4), (4, 5), (7, 8)\}$$

entonces después de hacer `apply_map(L,M,ML)`, debe quedar

$$\texttt{ML} = (2, 3, 4, 5, 8, 2, 3, 4)$$

Restricciones: No usar estructuras auxiliares. El tiempo de ejecución del algoritmo debe ser $O(n)$, donde n es el número de elementos en la lista (asumiendo que las operaciones usadas de correspondencia son $O(1)$).

- [**PROG-P2**]

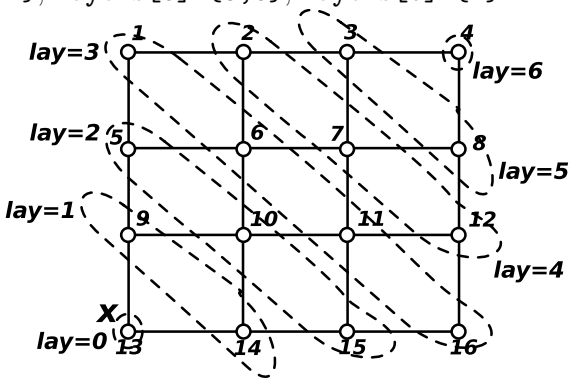
1. [**comptree**] Se define una relación de orden entre árboles binarios de enteros de la siguiente forma: `A<B` si `a<b`, o bien (`a=b` y `Ai < Bi`), o bien (`a=b` y `Ai=Bi` y `Ad<Bd`), donde `a`, `b` son las raíces y `Ai`, `Ad`, `Bi`, `Bd` son los subárboles izquierdos y derechos de `A` y `B`. Un nodo 'lambda' equivale a $-\infty$. **Consigna:** Escribir una función
`bool es_menor(tree<int> &A,tree<int> &B)` que retorna verdadero si `A < B`.
2. [**ord-nodo**]
Escribir una función predicado `bool ordnodo(tree<int> &A);` que verifica si cada secuencia de hermanos del subárbol del nodo `n` (perteneciente al árbol ordenado orientado `A`) estan ordenadas entre sí, de izquierda a derecha. Por ejemplo, para el árbol `(3 5 (6 1 3) (7 4 5))` debería retornar `true`, mientras que para `(3 9 (6 1 3) (7 4 2))` debería retornar `false`, ya que las secuencias de hermanos `(9 6 7)` y `(4 2)` NO están ordenados. Se sugiere el siguiente algoritmo: para un dado nodo retornar false si: 1) sus hijos no estan ordenados o 2) algunos de sus hijos contiene en su subárbol una secuencia de hermanos no ordenada (recursividad).

- [**PROG-P3**]

1. [**flat**]
Se está diseñando una red inerconectada por switches y se desea, para reducir lo más posible la *latencia* entre nodos, que cada par de nodos esté conectado en forma directa por al menos un switch. Sabemos que el número de nodos es `n` y tenemos un `vector< set<int> > sw` que contiene para cada switch el conjunto de los nodos conectados por ese switch, es decir `sw[j]` es un conjunto de enteros que representa el conjunto de nodos inteconectados por el switch j.
Consigna: Escribir una función predicado `bool flat(vector< set<int> > &sw, int n);` que retorna verdadero si cada par de enteros (j, k) con $0 \leq j, k < n$ está contenido en al menos uno de los conjunto en `sw[]`.
2. [**mklayers**]
Escribir una función
`void mklayers(vector<set<int> > &G, int x, vector<set<int> > &layers)` que dado un grafo `G` y un vértice de partida `x` determina la estructuras de capas de vecinos `layers` de `x` definida de la siguiente manera:

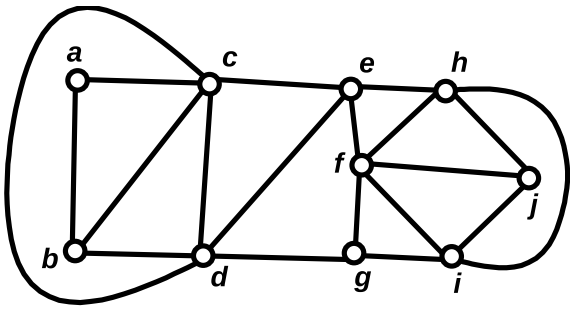
- La capa 0 es el conjunto `layers[0]={x}` .
- La capa 1 es el conjunto de los vecinos de `x`.
- Para `l>1` la capa `l` es el conjunto de los vecinos de los nodos en la capa `l-1` que no están en capas anteriores (0 a `l-1`). Notar que en realidad sólo hace falta verificar que no estén en las capas `l-1` y `l-2`.

Puede demostrarse que los vértices en la capa `l` son los que están a distancia `l` de `x`. Por ejemplo, dado el grafo de la figura, y partiendo del nodo `x=0` las capas son `layers[0]={0}`, `layers[1]={9,14}`, `layers[2]={5,10,15}`, `layers[3]={1,6,11,16}`, `layers[4]={2,7,12}`, `layers[5]={3,8}`, `layers[6]={4}`.



- **[OPER-P1]**

1. **[color-grafo]** Colorear el grafo de la figura usando el mínimo número de colores posible. Usar el algoritmo heurístico ávido siguiendo los nodos en el orden indicado $(a, b, c, ...)$. ¿La coloración obtenida es óptima? Justifique.



- **[OPER-P2]**

  - **[rec-arbol]** Dibujar el árbol ordenado orientado cuyos nodos, listados en orden previo y posterior son
    - `ORD_PRE` $=\{Z, A, R, Q, L, M, N, T, S, W, Q\}$,
    - `ORD_POST` $=\{Q, L, R, N, M, A, W, Q, S, T, Z\}$.
  - **[huffman]** Dados los caracteres siguientes con sus correspondientes probabilidades, contruir el código binario y encodar la palabra `CONCLAVE` $P(C) = 0.05, P(O) = 0.05, P(N) = 0.1, P(A) = 0.2, P(L) = 0.2, P(V) = 0.1, P(E) = 0.1, P(Q) = 0.2$ Calcular la longitud promedio del código obtenido.
  - **[part-arbol]** Particione el árbol AOO `(p (q (s x y) t u) (r v w))` con respecto al nodo `q`, es decir indique cuales son sus antecesores y descendientes propios, derecha e izquierda.

- **[OPER-P3]**

- [**heap-sort**] Dados los enteros $\{0, 4, 7, 1, 2, 12, 9, 3\}$ ordenarlos por el método de *"montículos"* (*"heap-sort"*). Mostrar el montículo (minimal) antes y después de **cada** inserción/supresión.
- [**quick-sort**] Dados los enteros $\{4, 8, 3, 0, 4, 9, 7, 2, 2, 1, 10, 5\}$ ordenarlos por el método de *"clasficación rápida"* (*"quick-sort"*). En cada iteración indicar el pivote y mostrar el resultado de la partición.
- [**abb**] Dados los enteros $\{16, 10, 23, 5, 6, 13, 8, 7, 4, 15\}$ insertarlos, en ese orden, en un *"árbol binario de búsqueda"*. Mostrar las operaciones necesarias para eliminar los elementos 12, 10 y 7 en ese orden.
- [**hash-dict**] Insertar los números $5, 18, 28, 11, 10, 38, 20, 7, 30$ en una tabla de dispersión cerrada con $B = 10$ cubetas, con función de dispersión $h(x) = x$ `mod` 10 y estrategia de redispersión lineal.

- [**PREG-P1**]

1. Explique que quiere decir la propiedad de *"Transitividad"* de $O()$.
2. ¿Puede tener una correspondencia varios valores iguales del dominio, o sea varias claves iguales? (Por ejemplo `M1={(1,2),(1,3)}`) ¿Y varios valores iguales del contradominio? (Por ejemplo `M2={(1,2),(3,2)}`)
3. ¿Qué retorna la dereferenciación de un iterator sobre correspondencias (clase `map`)?
4. ¿Porqué decimos que $(n+1)^2 = O(n^2)$ si siempre es $(n+1)^2 > n^2$?
5. ¿Qué ocurre si se invoca el `operator[]` sobre una correspondencia con una clave que no tiene asignación? Por ejemplo `M={(1,2),(3,4)}` y hacemos `x = M[5]`. ¿Da un error? ¿Qué valor toma `x`?

- [**PREG-P2**]

1. ¿Por qué no es útil definir a un "iterator"sobre un árbol ordenado orientado como `typedef *cell` como se hace en listas simplemente enlazadas?
2. Discuta si es posible insertar en una posición dereferenciable en ÁRBOLES BINARIOS.
3. ¿Para qué sirve el algoritmo de Huffmann para generar códigos? ¿En qué casos es bueno? ¿Da una solución *"heurística"* o da el código óptimo.
4. ¿Cómo se traduce la *condición de prefijos* para códigos representados por árboles? El árbol `(. (. a b) (c . d))`, ¿lo satisface?
5. Códigos de Hufmann: ¿De qué dependen las profundidades de los nodos en el árbol?

- [**PREG-P3**]

1. ¿Defina claramente cuál es al condición de *"Árbol Binario de Búsqueda"*?
2. ¿Por qué se les dice *"lentos"* a ciertos algoritmos de ordenamiento? Enumere los algortimos lentos que conoce.
3. ¿Cómo se encuentra el mínimo y el máximo de los valores en un árbol binario de búsqueda? ¿Cuál es la complejidad algorítmica de esta operación (caso promedio, mejor y peor)?
4. Discuta el valor de retorno de `insert(x)` para conjuntos.
5. Discuta el número de inserciones que requieren los algoritmos de ordenamiento lentos en el peor caso.